2010 Fifth International Conference on Internet and Web Applications and Services

The SHIP Validator: An Annotation-based Content-Validation Framework for Java Applications

Federico Mancini, Dag Hovland, Khalid A. Mughal
Department of Informatics
University of Bergen
Bergen, Norway
{*federico.mancini,dag.hovland,khalid.mughal*}*@ii.uib.no*

***Abstract*—In this paper, we investigate the use of Java annotations for software security purposes. In particular, we implement a framework for content validation where the validation tests are specified by annotations. This approach allows to tag what properties to validate directly in the application code and eliminates the need for external XML configuration files. Furthermore, the testing code is still kept separate from the application code, hence facilitating the creation and reuse of custom tests. The main novelty of this framework consists in the possibility of defining tests for the validation of multiple and interdependent properties. The flexibility and reusability of tests are also improved by allowing composition and boolean expressions. The main result of the paper is a flexible framework for content-validation based on Java annotations.**

***Keywords*-validation; annotations; java; content-validation**

I. Introduction

The OWASP Top Ten Project [1] lists the lack of proper input validation as the most prevalent cause of critical software vulnerabilities. For this reason, it is important to check that all input satisfies the criteria under which it is safe to execute the program. As an example, take a Java [2] program performing integer division. Integer division by 0 is an illegal operation, resulting in a runtime exception. Hence the value of the divisor should always be validated.

Carefully designing the application can alleviate problems caused by incorrect input. However, this alone will not prevent problems that might arise when a bad input is either passed on to other subsystems like databases, or manipulated and returned to the user.

Standard *input validation* mechanisms should make sure that all input is validated for length, type, syntax, and business rules before accepting the data to be displayed, stored or used [1]. This task can be repetitive and tedious for a programmer, and this is is the primary motive for implementing frameworks for input validation (Commons Validator [3], Struts 2 [4], Hibernate Validator [5] and Heimdall [6]). Such frameworks make it easier to maintain and execute the testing code by decoupling the application logic from the validation logic.

For object-oriented languages like Java, the challenge is to validate specific properties of an object representing the input, without writing validation code in the object itself. Historically, XML configuration files have been used to achieve this separation of concerns, by explicitly storing the names of the properties to be tested and that of the tests to be performed. At runtime, *reflection* [7] or Servlet filters (listener or interceptors) [4] would then used to actually run the tests on the target methods.

An alternate solution based on *annotations*, which were introduced in Java 5.0 [7], has gradually emerged. Approaches for input validation based on this new technology are described in [8]–[10], and employed, for instance, by Struts 2 [4] and Hibernate Validator [5].

Our approach is inspired by Heimdall [6], but adopts annotations instead of XML configuration, and provides more extensive and powerful tools for the creation of custom validation tests. The reasons to prefer annotations over XML configuration files have been well motivated in [9], and here we show in practice how far annotations can be pushed for input validation purposes. Although some technical solutions we use are also found in [8]–[10], we offer a simpler and more powerful way of creating custom tests, with focus on reusability. Furthermore, we propose a way of defining validation constraints over multiple properties of an object simultaneously, rather than just single properties. This allows the user to validate the relationship between interdependent properties, which, to our knowledge, is not possible with any other validation framework based on annotations or XML. For this purpose we distinguish between *property-tests* and *cross-tests*. A property-test is used for the validation of a single object property (for instance, JavaBean properties accessible through getter-methods), whereas a cross-test is concerned with constraints involving multiple properties.

A full version of this paper with more technical details can be found at [11], while the latest version of its implementation is available from [12]. The next section shows a simple example of how annotations can be used for validation. This running example is gradually extended to show more advanced features of our framework. A more formal description of how new annotations can be created and used is given in Section III. Finally, we compare our work to the other framework we mentioned previously in

978-0-7695-4022-1/10 $26.00 © 2010 IEEE
DOI 10.1109/ICIW.2010.26

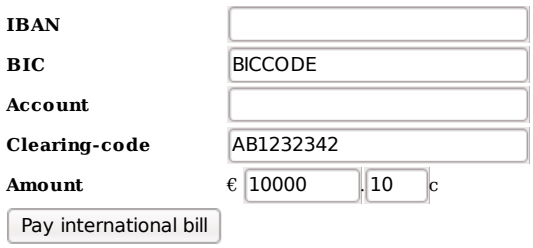

Figure 1: *Web form for international bank transfer.*

this section, and draw some conclusions.

II. A Running Example

In this section, we introduce the running example used throughout the paper, and show how annotations can be used to define tests on single properties of an object.

We will use the web form for international money transfers from a hypothetical Internet bank (see Figure 1). *IBAN* (International Bank Account Number) is the standard for identifying bank accounts internationally (not in USA). Some countries have not adopted this standard, and for money transfer to these countries, a special *clearing code* is needed in combination with the normal account number of the beneficiary. *BIC* (Bank Identifier Code), also known as SWIFT. It is needed to identify the beneficiary's bank uniquely.

We assume that the object representing the form is created in Java, and that each field in the web form is represented by a property of this object. Fields where the user does not enter a value, are in this example represented by the `null` value. A partial implementation of this Java object is shown in Figure 2. Here every annotation represents a test to be run on the return value of the method it is applied to. In our framework, annotations representing tests are called *validation-annotations*. This categorization is further split into *property-annotations*, which represent property-tests, and *cross-annotations*, which represent cross-tests. All the annotations in Figure 2 are property-annotations, i.e., they involve checking a single specific property.

We use property-tests to check whether basic formatting rules are respected. For example, the annotation `@IntRange(min=0,max=10000)` represents a test that checks whether the value of `amountEuro` is non-negative and not greater than 10000. The property-annotation `@IntRange(min=0,max=99)` represents a test to check whether `amountCents` is between 0 and 99. The property-annotation `@ValidateBIC` represents a property-test for BIC codes, and `@Required` means that the field cannot be left empty.

The annotations only specify what tests should be run on each value. To actually run the tests, an object must be passed to a validator. The validator inspects the object through reflection, extracts the annotations and the return

```
@ValidateBIC
@Required
public String getBIC()
 { return BIC; }

@IntRange(min=0,max=10000)
public Integer getAmountEuro()
 { return amountEuro; }

@IntRange(min=0,max=99)
public Integer getAmountCents()
 { return amountCents; }
```

Figure 2: *Example code using the property-annotations to test input from the web form in Figure 1.*

```
@Validation
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.ANNOTATION_TYPE,
         ElementType.METHOD})
public @interface IntRange {
 int min();
 int max();
 public static class Tester
 implements IPropertyTester<IntRange,
                            Integer> {
  public boolean runTest(IntRange r,
                         Integer v) {
   return(v >= r.min() && v <= r.max());
}}}
```

Figure 3: *Example of property-annotation.*

values from the getter-methods, and invokes the corresponding tests. This process is discussed in details in [11].

III. Validation annotations and tests

In this section we discuss the reasons for distinguishing between property-tests and cross-tests, and provide details of how they are implemented and used.

A. Property-annotations and Property-tests

Creating a property-annotation is fairly straightforward. As an example we use the declaration of `@IntRange` shown in Figure 3.

A fundamental part of the declaration is the meta-annotation `@Validation`, which works as a marker. Without it, our framework would not be able to distinguish a property-annotation from other annotations. There are other solutions to this problem, but such *marker-annotation*s are a standard way to compensate for the lack of inheritance in annotations [8]–[10].

The `@Retention` meta-annotation must be present such that the property-annotation is accessible at runtime. The annotation `@Target` has the usual

```
public interface IPropertyTester
 <A extends Annotation,I> {
  public boolean runTest(A an, I o)
  throws ValidationException;
}
```

Figure 4: *The interface for the classes implementing property-tests.*

meaning, and it will be clear in Sect. III-C why we need `ElementType.ANNOTATION_TYPE` besides `ElementType.METHOD`. The annotation declaration itself is fairly standard and can be annotated with any number of other annotations.

Finally, we require a public *inner class* which must contain the code of the property-test associated with this property-annotation. This class must implement the interface shown in Figure 4 in order to ensure that it provides the implementation of the method `runTest()`, which is invoked by the framework to run the test. Another possible approach for associating a test to an annotation is explained in [8].

The test corresponding to the property-annotation `@IntRange` is defined in the inner class `Tester`, as shown in Figure 3. The method `runTest()` is called by reflection and takes as parameters an instance of the annotation and the object to test (that is, the return value of the method). We allow only one inner class implementing `IPropertyTester` in the annotation declaration.

1) Handling `null` values: Many validation frameworks provide an annotation `@Required` which indicates that a certain property should not be `null` [4]–[6]. However, no annotation seems to be provided for handling a property that *can* be `null`.

To understand why this might be useful, let us assume that we allowed the field BIC in Figure 1 to be left empty by the user, i.e., the method `getBIC()` in Figure 5 was annotated with `NotRequired` instead. This means that a `null` return value from the method `getBIC()` should not lead to a `NullPointerException` being thrown during the validation process. To achieve this, either the test represented by `@validateBIC` must be able to correctly handle a `null` value in this situation, or the framework should prevent any test to be run when `BIC` has the value `null`. In the first case the burden of treating this special case is left to the programmer, who must consider the possibility that any test he or she designs might be run on a `null` value. However, tests are supposed to be reusable and cannot account for all possible ways of treating a `null` value in different situations. The same problem arises when, in the absence of a `@Required` annotation, the framework should decide how to interpret a `null` value, at the risk of masking a possible error or causing one.

To avoid these problems, we provide the

```
@Required
public String getBIC()
 { return BIC; }

@ExactlyOneNull
@NotRequired
public String getIBAN()
 { return IBAN; }

@ExactlyOneNull
@AllOrNoneNull
@NotRequired
public String getAccount()
 { return account; }

@AllOrNoneNull
@NotRequired
public String getClearingCode()
 { return clearingCode; }
```

Figure 5: *Examples of cross-annotations.*

`@NotRequired` annotation, which can be used to specify that a `null` return value is valid, and that in this case no other tests should be run on the value. If neither a `@Required` nor a `@NotRequired` annotation is specified, the framework will simply run the other tests on the method, even if the return value is `null`. Therefore, by default, our framework does not give any special treatment to `null` values, and the programmer can design reusable tests, by handling a `null` value in an independent way. In this setting, any `NullPointerException` will properly signal a programming error.

B. Cross-annotations

Recall the specifications of international bank transfers mentioned in Section II. All transfers require the BIC code of the receiving bank, and in addition either the IBAN or both clearing code and the account number. This means that there is a mutual dependency between some fields of the web form. Therefore, in order to check such constraints in the corresponding object, it is not enough to consider the return values of the involved methods independently. For this purpose we introduce a new type of validation-annotation which we have called cross-annotations. These allow a programmer to create tests involving multiple properties of an object, i.e., cross-tests.

In Figure 5, we extend the example shown in Figure 2 to show how it is possible to annotate the web form object in order to enforce the constraints mentioned earlier. Each cross-test is represented by a cross-annotation, which is applied to all methods whose return values are involved in the test. All annotations in the example are cross-annotations with the exception of `@Required` and `@NotRequired`.

The property-annotations from Figure 2 are not shown in order to keep the example simple.

The cross-test represented by `@ExactlyOneNull`, which is applied to the return values of `IBAN` and `clearingCode`, ensures that exactly one of them has not been filled in the web form. Furthermore, the cross-test represented by `@AllOrNoneNull` makes sure that either all or none of the methods marked with it return `null`. Thus, we are able to check that either the IBAN is used, or both the account number and the clearing-code are specified, but not all three.

Cross-annotations can be declared in almost the same way as property-annotations, as shown in Figure 6. Only the marker-annotation, `@CrossValidation`, and the interface of the inner test class, shown in Figure 7, are different.

```
@CrossValidation
public @interface AllOrNoneNull {
 public static class Tester implements
 ICrossTester<AllOrNoneNull,String> {
  public boolean runTest(
              AllOrNoneNull c,
              ArrayList<String> v) {
   ...
}}}
```

Figure 6: *Example of cross-annotation declarations.*

As can be seen in Figure 7, a cross-test takes as parameter the corresponding cross-annotation and all the return values involved in the test as a single `ArrayList`. This means that the return values are not differentiated according to what method they come from, hence limiting the type of cross-tests that can be developed in the current framework. These limitations are discussed in the next section.

C. Boolean composition

Another novelty of our approach is that we can combine validation-annotations with boolean operators in order to create new validation-annotations. These *composed* annotations can be created by declaring a new validation-annotation which is annotated with the validation-annotations we want to compose. In addition, the special meta-annotation `@BoolTest` can also be used in the composition. Its single element is of type `public enum BoolType{OR, AND, ALLFALSE}`, with the usual semantics. By default, specifying a list of annotations without the `@BoolTest` annotation represents the conjunction of the corresponding tests, thus `BoolType.AND` is not strictly necessary.

```
public interface ICrossTester
<A extends Annotation, V> {
    public boolean runTest
        (A a, ArrayList<V> v)
    throws ValidationException;
}
```

Figure 7: *The interface for cross-tests.*

```
@Validation
@BoolTest(BoolType.AND)
@PatMatch("\\w{8}|\\w{11}")
@AdditionalTest
public @interface ValidateBIC{}
```

Figure 8: *Definition of the annotation* `@ValidateBIC`*, used in Figure 2.*

```
@AmountCheck
public Integer getAmountEuro()
 { return amountEuro; }

@AmountCheck
public Integer getAmountCents()
 { return amountCents; }
```

Figure 9: *Examples of cross-annotations.*

Figure 8 shows the declaration of the annotation `@ValidateBIC` which we first introduced in Figure 2. This annotation is created by composing `@PatMatch("\\w{8}|\\w{11}")`, which is a common annotation for string-matching tests, and the one represented by `@AdditionalTest`, which represents some other possible test that we do not specify here. Since the annotation `@BoolTest(BoolType.AND)` is also specified, the test represented by `@ValidateBIC` will succeed only if *both* the tests represented by the two other property-annotations succeed.

Boolean composition can also be applied with cross-annotations. For example, if we in the web form above want to check that the overall amount transferred is greater than 0.00, but not greater than 10000.00, we can use the cross-annotation `@AmountCheck` as shown in Figure 9, since the fields for Euros and cents are represented as different properties in the web form object.

In Figure 10, we see that `@AmountCheck` is a composition of two other cross-annotations: `@SumMin(1)` and `@MaxAmount`. The first represents a test checking that the sum of `amountEuro` and `amountCents` is greater than 0. The second is in turn a composed cross-annotation, which by checking that either one of the two values is smaller than 1, or both are smaller than 10000, can guarantee that the total amount they represent together is at most 10000.00 (the logical structure is shown in Figure 12). This example also shows that composition is recursive. This allows the encapsulation of a complicated validation policy into a single annotation, thus improving readability, usage, and reusability.

```
// MaxAmount declaration
@CrossValidation
@BoolTest(BoolType.OR)
@OneLessThan(1)
@AllLessThan(10000)
public @interface MaxAmount {}

//AmountCheck declaration
@CrossValidation
@BoolTest(BoolType.AND)
@SumMin(1)
@MaxAmount
public @interface AmountCheck {}
```

Figure 10: *Composition of cross-annotations.*

Table I: *Examples of each of the four types of tests.*

	Basic Tests	Composed Tests
Property-Tests	`@IntRange`	`@ValidateBIC`
Cross-Tests	`@AllOrNoneNull`	`@AmountCheck`

Now that we introduced composition, it might become clearer why we imposed the limitations discussed in the previous section on cross-tests. If we allowed elements in cross-annotations which could be associated to a particular return value, for instance to identify the method they came from, composition would become more involved.

However, elements that are used to configure the test represented by the annotation, as in `@SumMin`, are allowed. Furthermore, since this parameter should be the same for each instance of the cross-annotation appearing in the object, it is a good practice to encapsulate the annotation with the specific element value into a new cross-annotation without any elements.

As a last observation, it might be convenient to use existing property-annotations to create cross-annotations. The reason is that the returned values involved in a cross-test are not differentiated by the method that generated them, and are often of the same type. The idea is that many cross-tests only check how many of the returned values have a certain property, or whether some combination, e.g., the sum, the product, or the average, has a certain property. In other words, either a single property-test is applied to a certain combination of the return values, or a property-test is applied to each return values and we count how many passed it. Hence, a cross-annotation can be constructed by combining property-annotations with special annotations that allow the user to define either the amount of return values which has to satisfy the property (all, none, at least n, exactly n), or an operator to combine them (addition, multiplication, concatenation, etc.). This has been implemented and more details are available in [11].

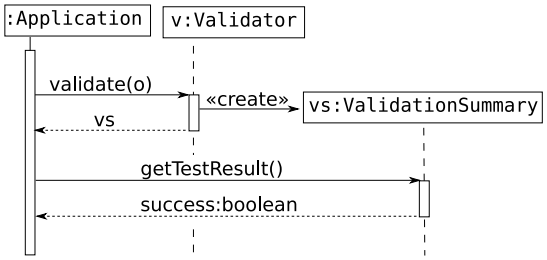

Figure 11: *Sequence diagram of the validation process.*

IV. Validation Summary

The technical implementation details of the framework can be found in [11]. Here we give only a high level description of the parts that the user interacts with.

In Figure 11, we show the sequence diagram of the validation process as it is seen by the user. Only a few lines of code needs to be inserted into the application in order to use the framework. Namely, a new `Validator` object has to be created, and its validation method `validate()` has to be invoked on an annotated object `o` to validate it. A `ValidationSummary` object is returned, containing the results of the validation tests for the object `o`.

A validation summary has a tree-like structure which mimics the boolean composition of the validation-annotations involved in the test (see figure 12). This is necessary due to the presence of boolean operators, as the failure of one single test is not enough anymore to declare that a property failed the validation. Only the boolean combination of the results of all partial tests can give the correct final answer. Since the set of tests responsible for the failure of the validation can be large and deeply nested, it is also possible to print out the content of the validation summary with the desired level of details, i.e., tree levels. An example printout of a validation summary is shown in Figure 13.

V. Related work

Mancini, Hovland, and Mughal [13] use the framework presented in the present paper to study the challenges and possibilities when using annotations for input validation.

There are other ways of tackling the input validation problem, but they are fundamentally different from our approach. For instance, there are static analysis tools [14] which provide support for tainting [15], [16] or tools that provide specific solutions for particular input validation vulnerabilities like the AntiSamy project for XSS [17].

Our framework is designed to allow the user to easily define and integrate custom validation tests in the application. Replacing the XML configuration files with annotations retains most of the advantages of having an external configuration file, like decoupling of validation logic from the application logic, and reusable tests. In addition, methods

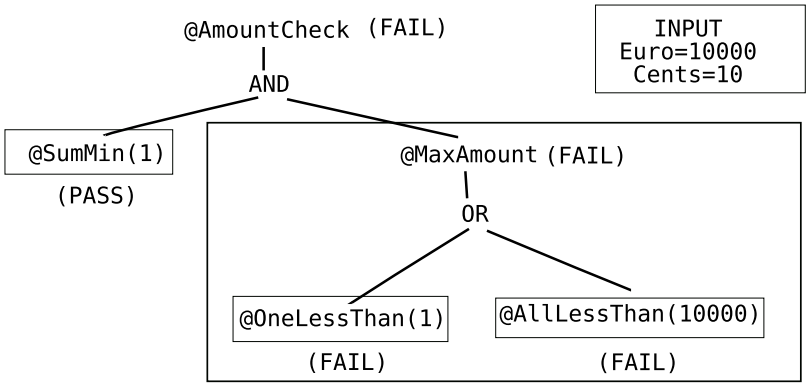

Figure 12: *Diagram showing the tree structure of a* `ValidationSummary` *object containing the validation results of the validation test corresponding to* `@AmountCheck` *on the input given in Figure 1.*

```
The value "BICCODE" returned by
"getBIC()" has not passed the
following property-test:
-Test: @ValidateBIC() because of:
 |-Test: @PatMatch(value=\w{8}|\w{11})
=====================================
The following cross-tests have failed:
-Test: @AmountCheck() because of:
 |-Test: @MaxAmount() because of:
   |-Test: @OneLessThan(value=1)
   |-Test: @AllLessThan(value=10000)
-Test: @AllOrNoneNull()
-Test: @ExactlyOneNull() because of:
 |-Test: @ExactlyNNull(value=1)
=====================================
```

Figure 13: *Printout of the* `ValidationSummary` *shown in figure 12.*

and classes do not need to be referred by string references any more, which is very error prone and requires additional maintenance. One disadvantage of switching to annotations, might be that runtime changes are not possible anymore. However, being forced to recompile after making changes helps to ensure the type safety of the application.

Additional advantages of using annotations instead of XML configuration files are discussed in Holmgren [9] and Hookom [8]. Both papers also include some technical solutions for using annotations to validate object properties, but only provide some basic illustrative code, rather than a fully functional framework. However, it seems like the ideas in [8] are the starting point for the work in [10], on which the the Hibernate Validator [5] is based.

Many basic technical solutions we use are similar to those provided in [8] and [10]. For example, using special meta-annotations as markers to allow the creation of custom validation-annotations and the way of associating tests to annotations. However, it must be said that these are standard solutions when annotations are involved.

When it comes to running the actual validation, we are close to the solutions proposed in [8], [10], which allow complete decoupling between validation and application code. In contrast, the solution in Holmgren involves inserting extra code inside the method to be validated. Although this approach allows tests on methods without return values, i.e., setter methods or methods with parameters, it makes the test code and the application code more interdependent, which is what we have tried to avoid.

Composition is also proposed in [10], but only conjunction of annotations is considered. In this case composition is simply a way of collecting annotations together, not of creating new constraints.

What is called a multi-valued constraint in [10], i.e., applying the same annotation with different element values to the same property, can easily be achieved in our framework by encapsulating each instance in another annotation as in Figure 3.

Struts 2 [4] and Stripes [18] also provide validation through annotations. Both frameworks offer a limited set of standard annotations, with no possibility of creating custom tests. As new annotations cannot be created, composition is not possible and the only way to add custom tests is to use a `@CustomValidator` annotation which takes as argument the name of the test. This is then associated to the corresponding class in an XML configuration file. In other words, despite the use of annotations, classes are still referenced by string names.

Most importantly, none of the mentioned related work seems to consider the possibility of validating multiple properties. We consider our cross-annotations a natural extension of validation-annotations which can add expressive power to the validation-tests that the user can design. Besides, we manage to keep most of the technicalities involved in cross-validation hidden inside the framework, so that there is almost no difference between property- and cross-annotations from a user point of view, and usability is not

compromised.

Finally, most of these frameworks are mainly designed to work with JavaBeans, and make strong assumptions about the type of applications that can utilize them. Our framework, as Heimdall, does not assume much about the application, and should be easy to integrate with any Java project.

VI. Conclusion and future work

We have implemented a flexible content-validation framework [12] based on Java annotations, which can easily be integrated into existing applications. The main idea in the design of this framework has been that it should be easy to create libraries of custom validation-annotations, and that these tests should be highly reusable. We have tried to provide simple, yet powerful means for doing this, for example, using boolean composition. Besides, we have pushed the limits of annotations by allowing constraints involving interdependent properties, which have not been addressed in any work we are aware of.

For future work, we intend to extend the library of predefined annotations by creating new tests aimed at specific input validation vulnerabilities and improve the validation summary to support specific queries.

References

[1] (2009, May) OWASP Top Ten project. [Online]. Available: http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[2] (2009, September) Java 6. Sun. [Online]. Available: http://java.sun.com/javase/

[3] (2009, May) Commons validator. Apache. [Online]. Available: http://commons.apache.org/validator/

[4] (2009, May) Struts. [Online]. Available: http://struts.apache.org

[5] (2009, September) Hibernate validator. Hibernate. [Online]. Available: https://www.hibernate.org/412.html

[6] L.-H. Netland, Y. Espelid, and K. A. Mughal, "A reflection-based framework for content validation," in *ARES*. IEEE Computer Society, 2007, pp. 697–706.

[7] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language, Fourth Edition*. Addison-Wesley, 2006.

[8] J. Hookom, "Validating objects through metadata," O'Reilly, January 2005. [Online]. Available: http://www.onjava.com/lpt/a/5572

[9] A. Holmgren, "Using annotations to add validity constraints to javabeans properties," Sun, March 2005. [Online]. Available: http://java.sun.com/developer/technicalArticles/J2SE/constraints/annotations.html

[10] E. Bernard and S. Peterson, "Jsr 303: Bean validation," Bean Validation Expert Group, March 2009. [Online]. Available: http://jcp.org/aboutJava/communityprocess/pfd/jsr303/index.html

[11] D. Hovland, F. Mancini, and K. A. Mughal, "The SHIP validator: An annotation-based content-validation framework for java applications," Department of Informatics, University of Bergen, Tech. Rep. 389, September 2009.

[12] ——. (2010, February) SHIP validator. [Online]. Available: http://shipvalidator.sourceforge.net

[13] F. Mancini, D. Hovland, and K. A. Mughal, "Investigating the limitations of java annotations for input validation," in *SecSE*. IEEE Computer Society, 2010, in press.

[14] B. Chess and J. West, *Secure programming with static analysis*. Addison-Wesley Professional, 2007.

[15] W. Pugh, "Jsr 305: Annotations for software defect detection," September 2006. [Online]. Available: http://jcp.org/en/jsr/detail?id=305

[16] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," in *ACSAC*. IEEE Computer Society, 2005, pp. 303–311.

[17] (2009, May) OWASP AntiSamy project. OWASP. [Online]. Available: http://www.owasp.org/index.php/Category:OWASP_AntiSamy_Project

[18] (2010, February) Stripes framework. [Online]. Available: http://www.stripesframework.org